WinFS Synchronization

Lev Novik
Development Lead

Nils Pohlmann
Program Manager

03/30/04

WinFS

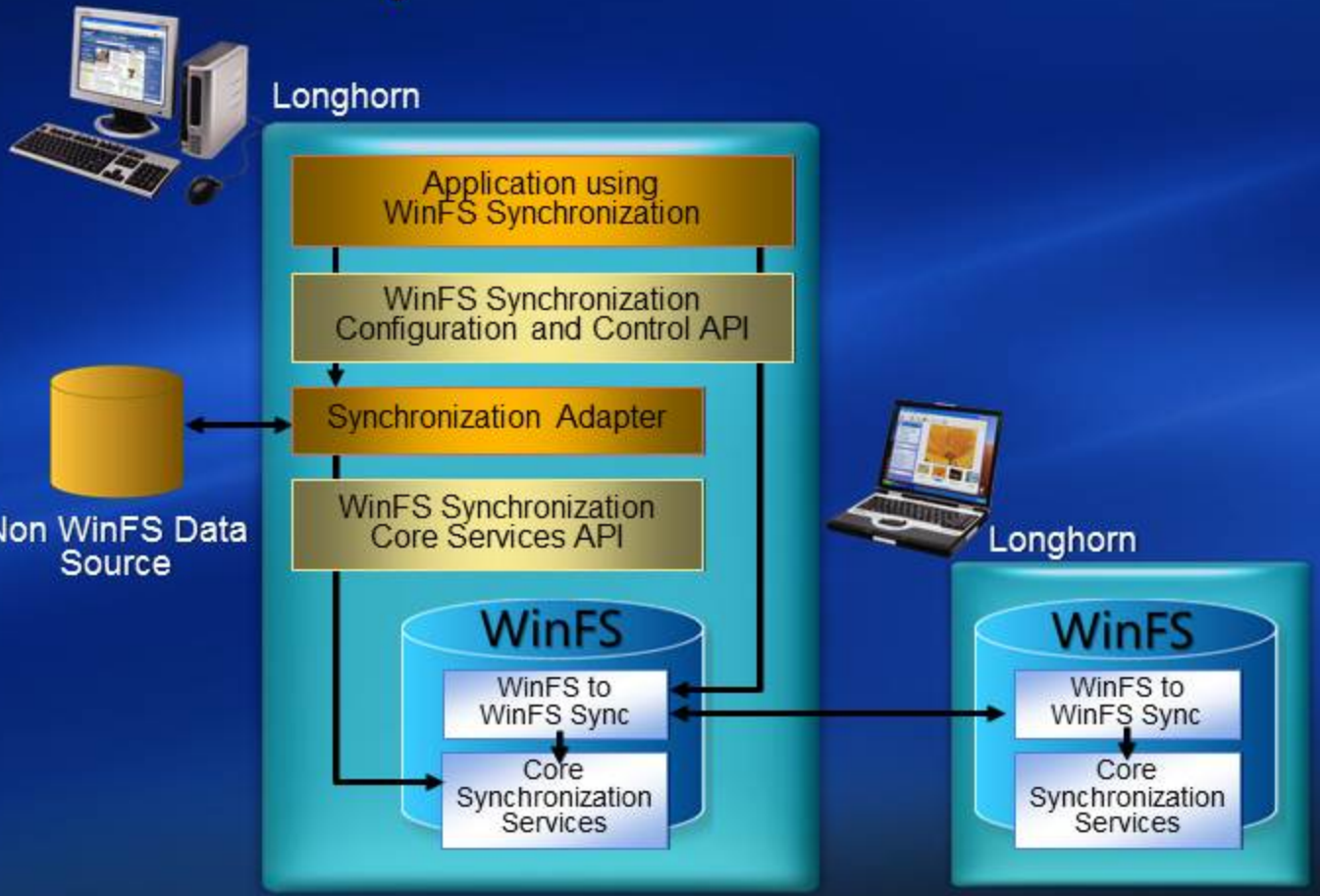
WinFS Synchronization Architecture

WinFS

Requirements and Approach

- Multi-master synchronization
  - Solve general case, single-master falls out
  - Support Peer-to-peer scenarios: set of peers is not known
  - All changes are sent to each replica exactly once (regardless of topology)
- Net-changes synchronization (state-based)
  - Send the end-result of all the operations since last sync
    - No need to send individual operations
  - Maintains bounded meta-data (small amount per item)
    - No need to maintain lists of all operations
- Sync at the level of WinFS rather than SQL
  - All enumeration is done through WinFS views
  - Changes are applied through update-grams
  - Sync all WinFS items: file-backed and non-file-backed
  - Operates in terms of WinFS organizational concepts:
    - Item Domain DAGs
    - Compound item trees

WinFS Sync Architecture



Core Synchronization Services

- Incremental Synchronization
  - Change Tracking
  - Change Enumeration
  - Change Application
- Conflict Management
  - Detection
  - Auto resolution
  - Logging
  - Programmatic resolution

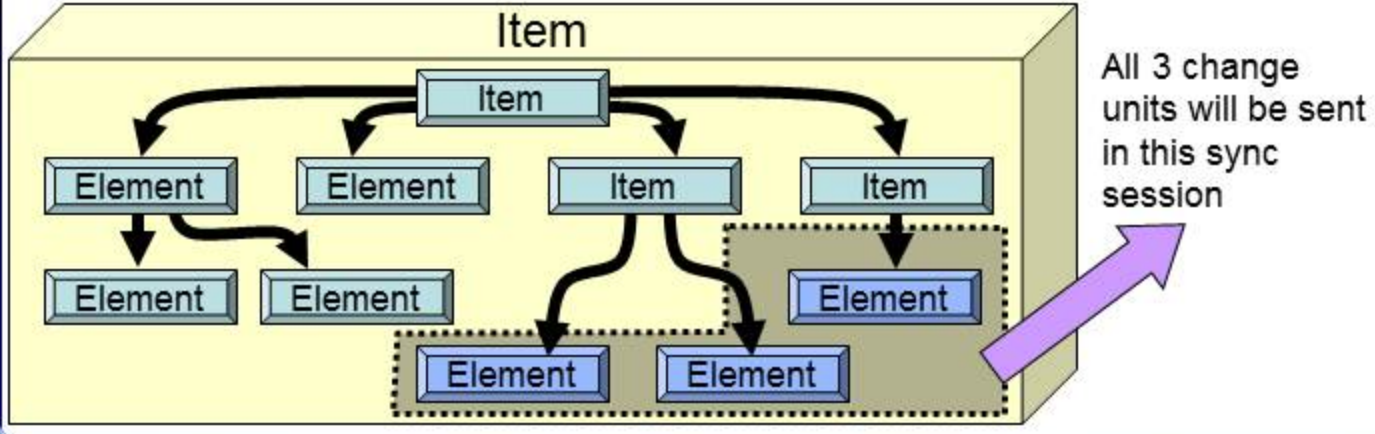
Change Tracking

- WinFS records change tracking information in every item
  - Maintains versions and tombstones through all operations
  - Common service for Sync, Notifications, and all WinFS applications
  - Occurs at the level of change units

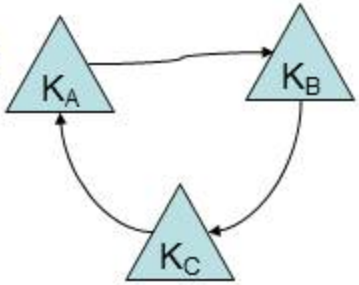
- Change units are the granularity of:
  - Incremental update propagation
    - only units that changed will be transmitted over the wire
  - Conflict detection
    - Two changes are only in conflict if independent (i.e. concurrent) updates are made to the same change unit

Consistency Boundary: Compound Items

- Consistency Boundary is a WinFS Item (+ embedded items +extensions)
  - If several change units were modified within an item since the last sync, you either get all of them or none of them
  - Parent/child relationships are preserved
  - Sync resumption: can interrupt at any item boundary and resume from the same place



Basic concepts: How Sync does it

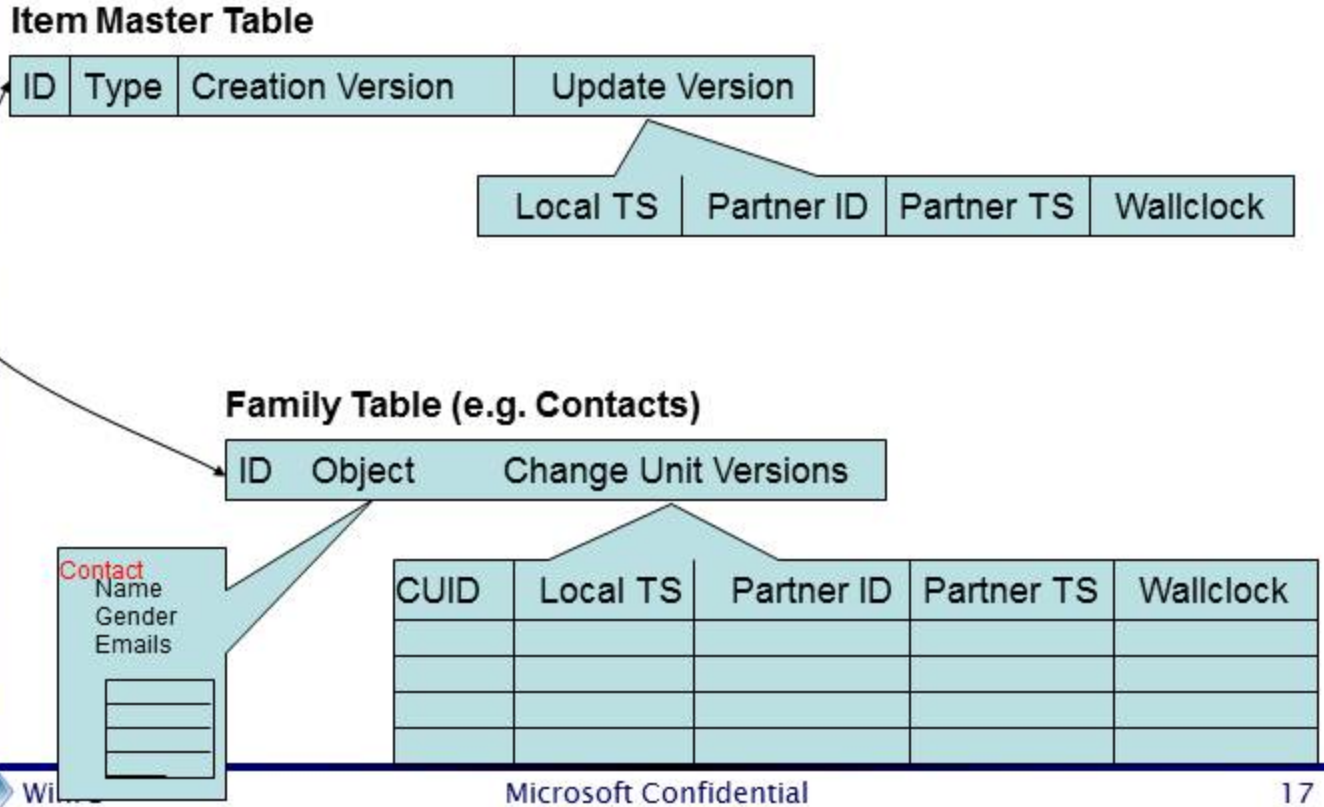


- Replicas are defined as folder hierarchies
- Each replica maintains its "knowledge"
  - Describes what the replica has already seen
  - Not pair-wise, understood by all replicas
- When enumerating changes:
  - A sends B his knowledge
  - B sends A only the *change units* that are not covered by A's knowledge
- When detecting conflicts:
  - Two changes conflict if and only if they where made without *knowledge* of each other
  - A declares a conflict if A's current version is not covered by B's knowledge
- Avoids the need to maintain version histories

Change Tracking Implementation

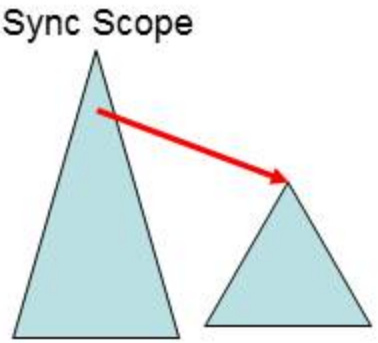
- Version syntax:
  <replica ID, replica version, local version>
- Change tracking maintains version info:
  - Per-entity (item, extension, relationship)
    - Creation/deletion and last-update version
    - Indexed
  - Per change-unit (as a single UDT)
    - Last-update version and wallclock
    - Not indexed!
- Tombstones:
  - Per entity, with deletion version
  - Cleaned up according to system-wide policy

Change tracking schema layout



Change Enumeration

- **Is given** a folder and the caller's knowledge
- **Finds** items that moved in
  - Tricky: a single link creation can move in a lot of items
- **Finds** items that were deleted or moved out
  - No difference to sync: no longer in the scope
- **Finds** change units that have been modified
  - First find modified entities using entity versions
  - Then look inside for modified change units
  - No need for change unit versions to be indexed



- **Returns** all of the above and the new knowledge
  - Highly heterogeneous dataset (use UDTs)
    - Unchanged units do not leave the process
  - Uses snapshot isolation (except for FileStreams)
  - Groups by whole compound items (to enable incremental application)

Change Application

- Is given:
  - A batch of changes
  - Caller's current knowledge
  - Recipient's knowledge increment
- Does:
  - Conflict detection by comparing to local version
  - Actual application if not in conflict
    - Using Updategrams; change-tracking is suppressed
    - Updates local Knowledge if successful
  - Call Conflict Manager if in conflict (below)

Conflict Management: Detection

- To determine if two changes conflict, a replica must determine if
  - (1) the semantics of the change operations are incompatible
    - Overlap-based (affecting the same change unit)
    - Constraint-based (applying both would violate a constraint)
      - Creating two files in the same folder with the same name
  - (2) the changes were performed concurrently (without knowledge of each other)
- An incoming change is in conflict with the local state if the last version of the local item is not covered by the knowledge of the change author
- An incoming change is obsolete if the incoming change is covered by local knowledge

Automatic Conflict Resolution

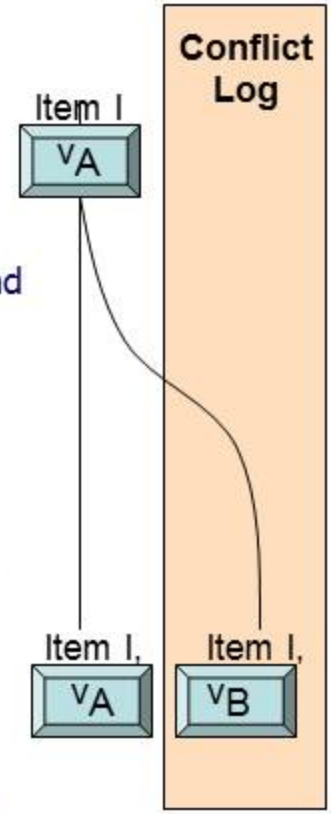
- Occurs synchronously, immediately after detection
- Conflict Handler is chosen according to policy and conflict information
- Conflict Handlers choose one of the following *outcomes*:
  - Local wins. Keep local data, disregard incoming change.
  - Remote wins. Overwrite local data, as if no conflict detected.
  - Propose new. Offer a new change that encompasses both
    - May affect items other than the one in conflict
  - Reject. Let the conflict be logged or resolved by somebody else
- Conflict resolutions propagate to other replicas
  - Identical-value resolutions do not cause additional conflicts
  - Different resolutions are arbitrated deterministically --- automatic convergence guarantee in unmanaged environments

System Conflict Handlers

- Local wins/remote wins
- Last-writer-wins (based on system clock values)
- Deterministic (based on replica ID)
  - Guarantee of convergence
- For name-collision conflicts
  - Rename
    - Renames one of the colliding items to a similar unique name
      - E.g. "Foo.1"
  - Merge
    - Merges the two items, effectively deleting one
    - Particularly useful for folders --- contents are merged
- Conflict filters route a conflict differently depending on:
  - Conflict type (e.g. Update-update, Update-delete, etc)
  - Item type (e.g. Contacts vs. Documents)
  - Identical values (special handling for concurrent identical changes)

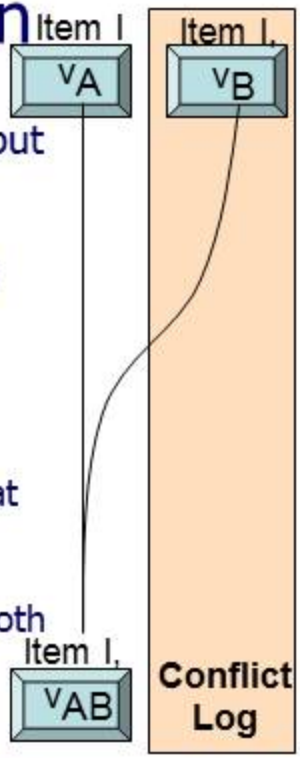
Conflict Logging

- Conflict logger is a system conflict handler
- Shared conflict log with SyncMan for all SyncMan-controlled synchronization
  - Base WinFS Schema, enough for SyncMan to display and route to handlers
  - WinFS-specific extensions.
- Conflicts are logged as independent items
  - Not inside the items they refer to
  - Each record contains:
    - The text of the remote change (the changed values) with their versions
    - The made-with knowledge of the remote change
    - The Item ID of the local item that is in conflict
    - Local configuration in effect at detection time
  - Conflict viewing can be done using regular WinFS APIs



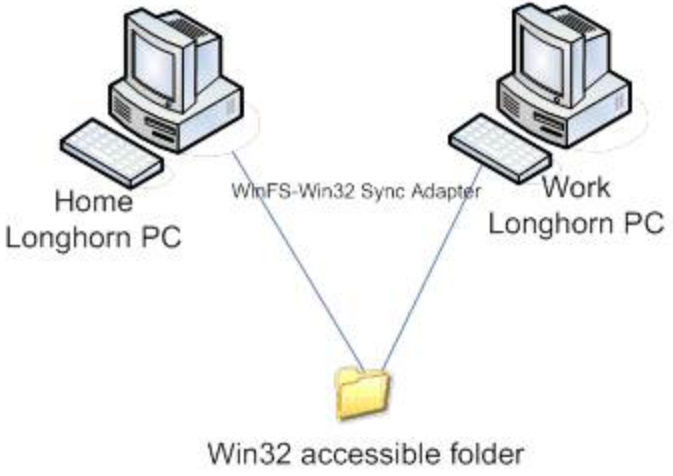
Programmatic Conflict Resolution

- Performed by a local application, often with Human input
  - Uses Conflict Inspection and Resolution API
- App provides a resolution, like a Conflict Handler
  - Local wins, remote wins, suggest new change supported
  - If a new conflict is detected due to a concurrent update
    - Resolution fails, conflict returned to app
- Applications cannot modify or delete conflict records
  - Except by using conflict resolution API.
  - Local updates are assumed to be made without looking at the log, hence not a resolution.
- Conflict records are automatically deleted
  - Upon receipt of a change made with the knowledge of both
  - Upon logging of a superseding conflict (made with knowledge of the remote change that was in conflict)
- Conflict resolutions propagate to other replicas
  - Identical-value resolutions do not cause additional conflicts
  - Different resolutions raise a new conflict to be handled



WinFS-Win32 Sync Adapter Demo

- Sync my stuff from my Home machine with my Work machine
  - Demo uses Home, Work and roaming folders on single PC



- Working WinFS-Win32 adapter from latest Lab06 build

Demo

Demo – Highlights

- WinWin adapter built on WinFS Sync APIs
- Heterogeneous collection of WinFS items
  - File-backed item sync
- Synchronization profile for session configuration
- WinFS change tracking and Core Sync RT Usage
  - Knowledge, Efficient Change enumeration and application
- Conflict Handling
  - Conflict Detection
  - Conflict Logging
  - Resolution by policy
- WinFS – Win32 adapter intermediary storage format

Annotating Schemas for Synchronization

WinFS

Declaring Change Units

- Change Units are declared in WinFS Schemas
  - Changing change unit granularity is a schema change
- Top-level elements of items, relationships, and extensions are grouped into Change Units
  - Highest granularity: a single *top-level* attribute of an item type
  - Nested elements cannot be split across change units
  - Schema Designer might choose less granularity -- configurable declaratively in the schema as groups of fields.
  - Smaller than a UDT!

Sync Change Units

```
<ItemType Name="SiteReport" BaseType="WinFS.Item" TypeId="…" >
    <ChangeUnit Name="BaseCU" Id="1"/>
    <ChangeUnit Name="PhotosCU" Id="2"/>
    <Property Name="Name" Type="WinFS.String" Size ="255" Nullable="false"
    ChangeUnit="BaseCU" />
    <Property Name="Date" Type="WinFS.DateTime" Nullable="false"
    ChangeUnit="BaseCU" />
    <Property Name="Location" Type="WinFS.String" ="255" Nullable="true"
    ChangeUnit="BaseCU" />
    <Property Name="Client" Type="WinFS.String" Size="255" Nullable="true"
    ChangeUnit="BaseCU" />
    <Property Name="Status" Type="WinFS.String" Size="255" Nullable="false"
    ChangeUnit="BaseCU" />
    <Property Name="Notes" Type="WinFS.String" Size="4000" Nullable="true"
    ChangeUnit="BaseCU" />
    <Property Name="Photos" Type="MultiSet" MultisetOfType="Photo"
    Nullable="true" ChangeUnit="PhotosCU" />
</ItemType>
```

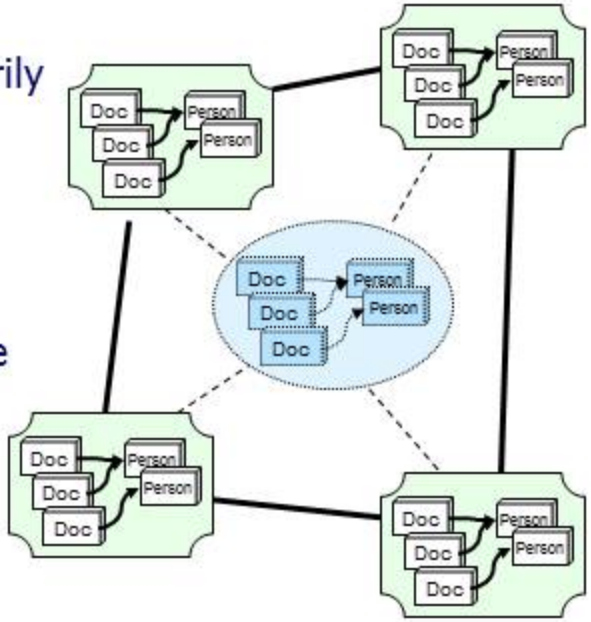
```
<NestedType Name="Photo" BaseType="WinFS.NestedType" TypeId="…" >
    <Property Name="Image" Type="WinFS.Binary" Size="max" Nullable="false"/>
    <Property Name="Caption" Type="WinFS.String" Size="255"
    Nullable="false"/>
</NestedType>
```

Additional Considerations

- Derived types can extend base type change units
  - Has the option of defining new change units
  - Has the option of adding new properties to old change units
- Design tradeoffs
  - Making change units too big leads to
    - Too much traffic on the wire (the whole change unit is sent every time anything inside it changes)
      - Isolate "big" fields individually from the others
    - Too many conflicts (every time two changes are made inside a change unit)
      - Isolate "rapidly-changing" fields from each other
  - Making change units too small leads to
    - Storage overhead (~40 bytes per change unit per item)
    - Update performance degradation
      - Exact impact to be measured

Sync Configuration Concepts

- **Sync Community**
  - Peers interested in synchronizing a given shared folder
  - The total set of peers is not necessarily known
- **Mappings**
  - Describes how shared objects are mapped to local objects
  - Every peer in the community has one
- **Profiles and schedules**
  - Define when and how to perform synchronizations between peers
  - May be persistent or ad-hoc
  - Any number per replica: sets up topology



Sync Configuration - mappings

- One per-replica per-community
  - Controls what to do upon receipt of a remote sync request
- Local sync endpoint
  - Local volume and replica root item ID
- Conflict Resolution Policies
- Community ID
- Replica ID
- More information on mappings and mapping manager can be found at: \\winfsbuild\specs\sync\specs\WinFS-WinFS Sync Configuration.doc

Sync Configuration: profiles

- Local mapping reference: what to sync
- Remote endpoint (URL): with whom to sync
- Direction (send, receive, both)
- Adapter Configuration :
  - which adapter to use, additional parameters if required
- Filters
  - what to send
    - e.g. "only Important contacts"
  - what to request
    - e.g. "only headers"
- Conflict resolution policies

- Spec for this is at: http://winfs/specs/sync/specs/WinFS Sync Controller API.doc

Sync Config & Control APIs

- Create a replica
- Request synchronization for a replica pair
- Cancel an active synchronization request
- Subscribe to events for
  - Synchronization start
  - Synchronization end
  - Progress status information

- Note on topology configuration
  - WinFS Sync stops at pair-wise configuration
  - Others (P2P, AD, Castle) are in charge of overall determination of topology
  - Convey their decisions to Sync by way of creating Sync Profiles

Create Replica

```
// This function is run on both machines where sync needs to be configured,
// with the correct path to the folder to be synchronized
//
private void CreateReplica(
            string rootFolderPath,
            string communityIdentifier )
{
    ItemContext ic = ItemContext.Open(rootFolderPath))
    try
    {
        Folder replicaRootItem = (Folder) ic.FindItemByPath(rootFolderPath);

        // Initialize the mapping
        WinfsSynchronizationMapping mapping =
            new WinfsSynchronizationMapping(communityIdentifier);

        mapping.ReceiveConflictPolicy =
            new ConflictResolverConfiguration(
                    ConflictResolverResolutionTypes.LastWriterWins);

        mapping.ReplicaRootItemId = replicaRootItem.ItemId;

        // Create the mapping
        WinfsSynchronizationMappingManager mappingMgr =
            new WinfsSynchronizationMappingManager(rootFolderPath);
        mappingMgr.SaveMapping(mapping);
    }
    catch …
}
```

Synchronize Replicas (1)

```
// Once the CreateReplica() function runs on both machines,
// the following function can be invoked on either machine
// to get the sync done
//
private void Sync()
{
    try
    {
        String communityIdentifier = "myTestCommunity";
        String remoteHostName = "myHost2";

        // Get the mapping
        //
        WinfsSynchronizationMappingManager mappingMgr =
            new WinfsSynchronizationMappingManager();
        WinfsSynchronizationMapping mapping =
            mappingMgr.GetMapping(communityIdentifier, null);

        // Get the profile from the mapping
        //
        SynchronizationProfile profile =
            mappingMgr.CreateMappingProfileForHost(
                mapping, remoteHostName, null,
                SynchronizationType.SendAndReceive);
```

Writing Synchronization Adapters

WinFS

Synchronization Adapter Responsibilities

- Outbound from WinFS
  - Ask WinFS Sync to enumerate changes in WinFS store
  - Transform data and apply to external store
  - Resolve conflicts (or defer to in-bound below)
- Inbound to WinFS
  - Enumerate changes in external store
  - Transform data and ask WinFS Sync to apply to WinFS store
  - WinFS Sync will resolve conflicts
    - May need help from the user

Change Enumeration (outbound)

- Adapter maintains the *knowledge* of the last synchronization --- what changes have been applied at the backend
  - Takes the format of WinFS Knowledge object
  - This object is opaque to the adapter
  - Should be stored on the server, if possible
    - Resilient to backup/restore, multi-master sync, etc
  - If not possible, can be stored in WinFS
- WinFS returns new, changed, and deleted items
  - Complete objects are returned (not individual change units)
  - Change units are marked to allow adapter to identify actual changes
  - All "echos" are suppressed: adapter's own changes are not returned

Change Enumeration: acknowledgement

- Adapter acknowledges changes once they are applied on the server
  - Callback interface is provided
- At the end of synchronization session, WinFS returns new *knowledge*
  - Covering all changes that were enumerated and acknowledged
  - Should be stored by the adapter for the next change enumeration

Change Enumeration Sample

```
ItemContext ctx = ItemContext.Open ( "\.\System\UserData\ashah\My  Contacts", true );

// Get the replica item id and remote partner id from the profile
Guid replicaItemId = SynchronizationProfile.LocalEndpoint.ReplicaItemId;
Guid remotePartnerId = SynchronizationProfile.RemoteEndpoint.RemotePartnerId;

// Lookup stored knowledge in the store
ReplicaKnowledgeItem knowledgeItem = ...; // Lookup correct item using ctx
ReplicaKnowledge remoteKnowledge = knowledgeItem.ReplicaKnowledge;

// Initialize ReplicaSynchronizer
ctx.ReplicaSynchronizer = new ReplicaSynchronizer( replicaItemId, remotePartnerId );
ctx.ReplicaSynchronizer.RemoteKnowledge = remoteKnowledge;
ChangeReader reader = ctx.ReplicaSynchronizer.GetChangeReader();
```

(contd. on next slide)

Q/A ?